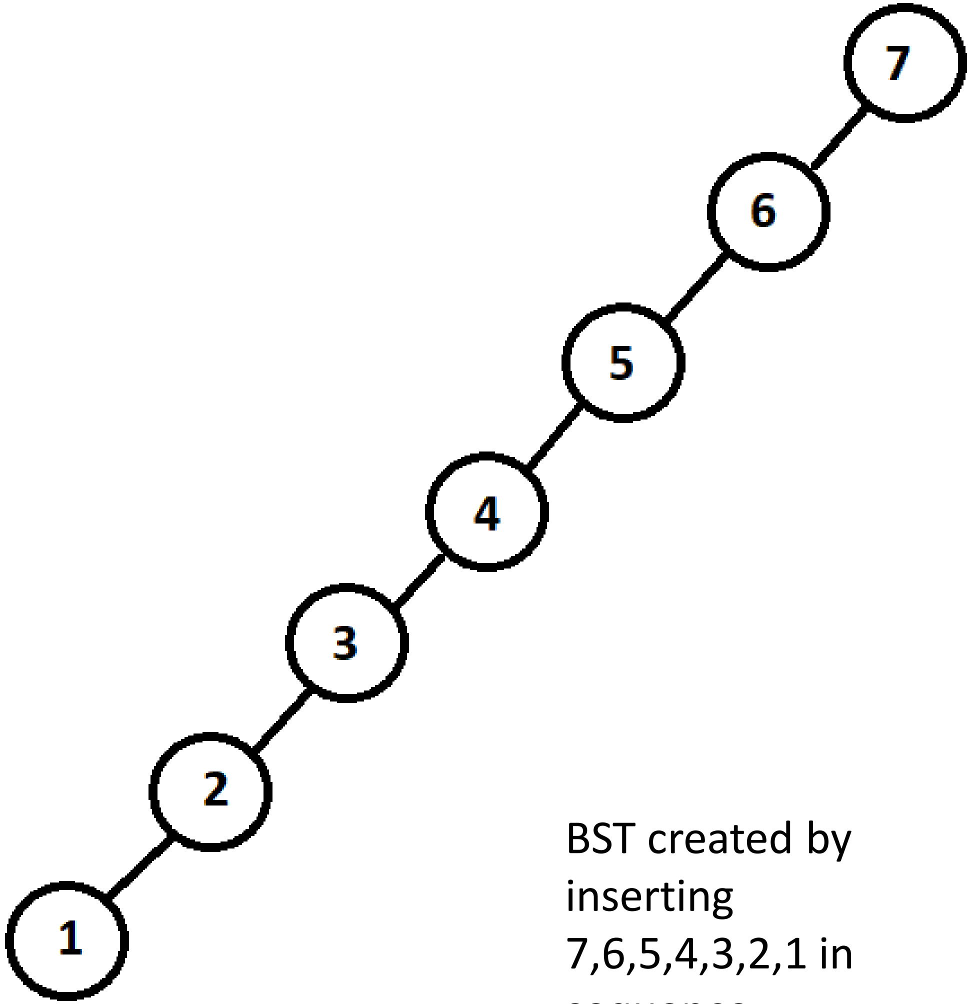
AVL trees

Motivation for AVL tree

- Worst-case BST is a linked-list
- Tree depth is $O(n)$ in worst-case
- Results in $O(n)$ time complexity for operations like `findMin`, `insert`, `delete`, etc.
- Want to avoid this worst-case behavior



BST created by inserting 7,6,5,4,3,2,1 in sequence

Balance condition

- Need to enforce a ‘balance condition’ on the tree that will rearrange tree when it becomes unbalanced, forcing it’s depth to be $O(logn)$
- Reminder: height of a tree is the length of the longest path from the root to a leaf
- Height of an empty tree is defined to be -1
- Idea 1: require left and right subtrees of root to have equal height
    - Not a good solution, can end up with a tree of height $n/2$ which is $O(n)$
- Idea 2: require that *every node* have left and right subtrees of equal height
    - Not a practical solution, because as soon as we add a node to the tree it violates this condition (can never add new nodes to the tree)

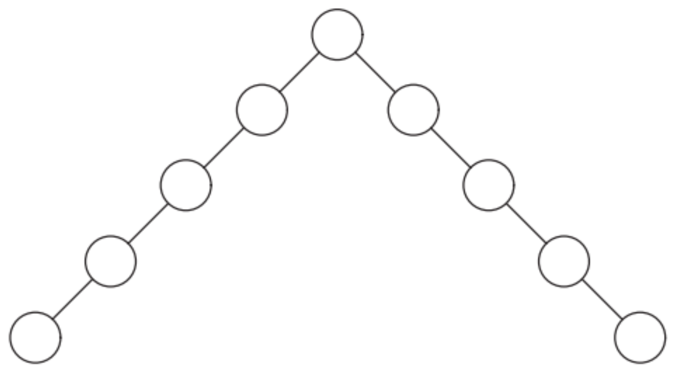
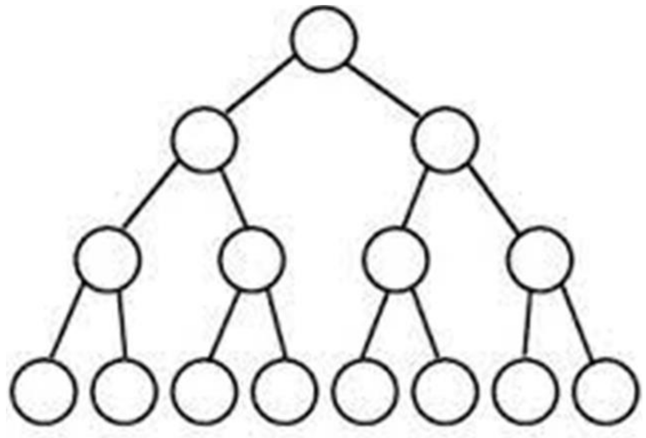


Figure 4.31 A bad binary tree. Requiring balance at the root is not enough.



AVL tree

- AVL (Adelson-Velskii and Landis) is a BST, but with the following balance condition:
- For every node in the tree, the height of left and right subtrees can differ by *at most 1*

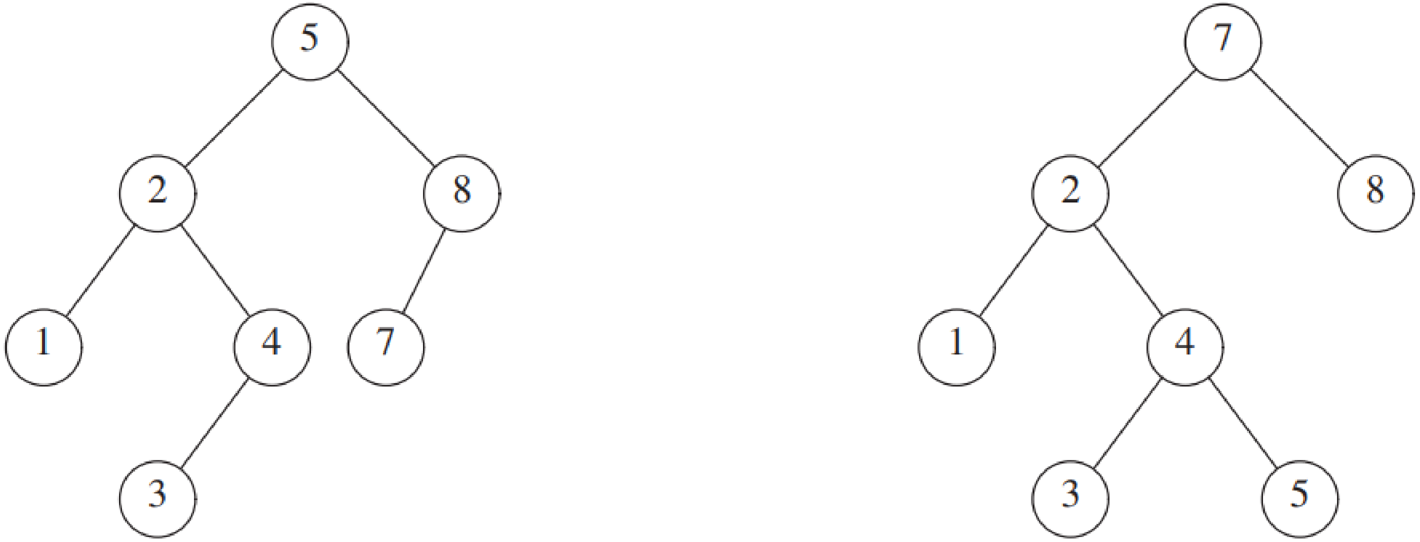


Figure 4.32 Two binary search trees. Only the left tree is AVL.

Smallest possible AVL tree of height 9

- AVL tree condition will ensure that height of tree is $O(logn)$
- All operations (`findMin`, `insert`, `remove`, etc.) can be completed in $O(logn)$ time
- Challenge is updating tree structure when we do an insert/remove to ensure AVL property is maintained

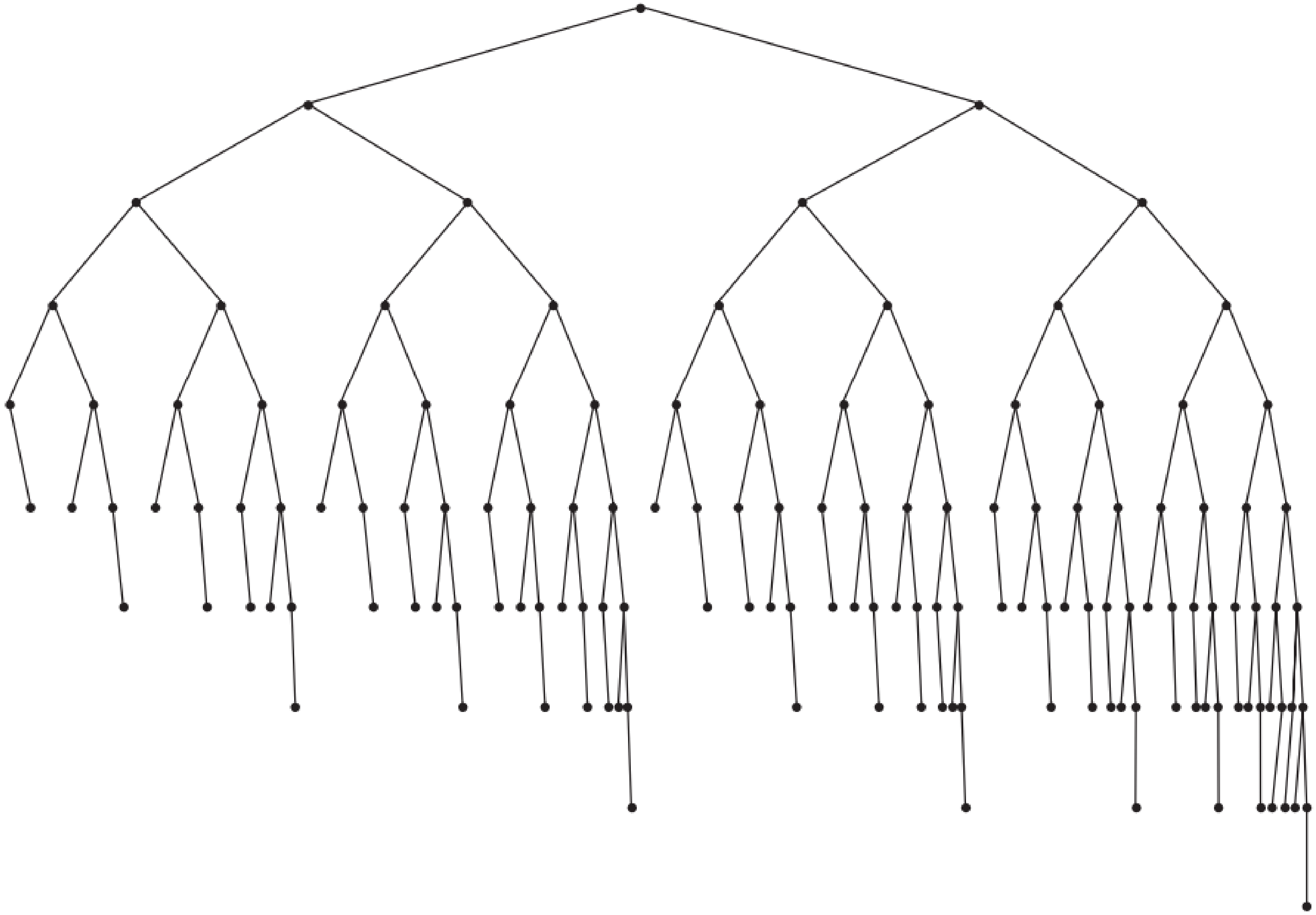


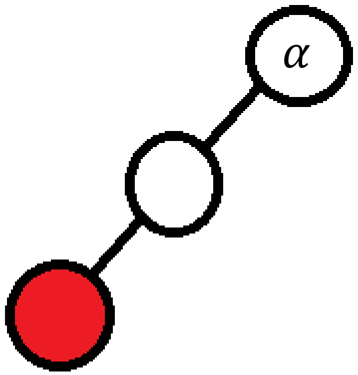
Figure 4.33 Smallest AVL tree of height 9

Contains 143 nodes

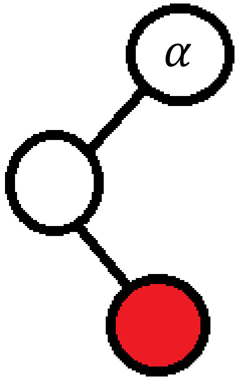
When does tree become unbalanced?

- Call the node that becomes unbalanced α
- Reminder: height of empty subtree = -1
- Four cases that can cause tree rooted at α to become unbalanced
- 1) insertion into left subtree of left child of α
- 2) insertion into right subtree of left child of α
- 3) insertion into left subtree of right child of α
- 4) insertion into right subtree of right child of α

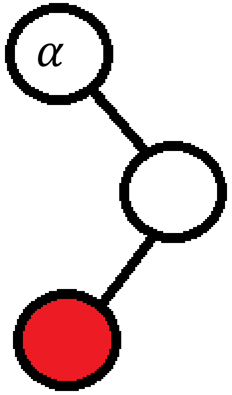
1)



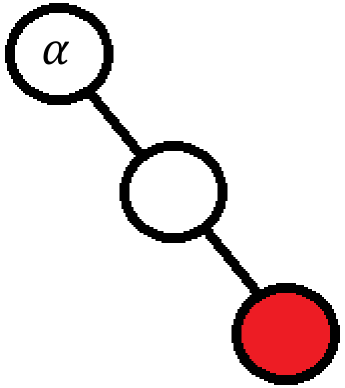
2)



3)



4)

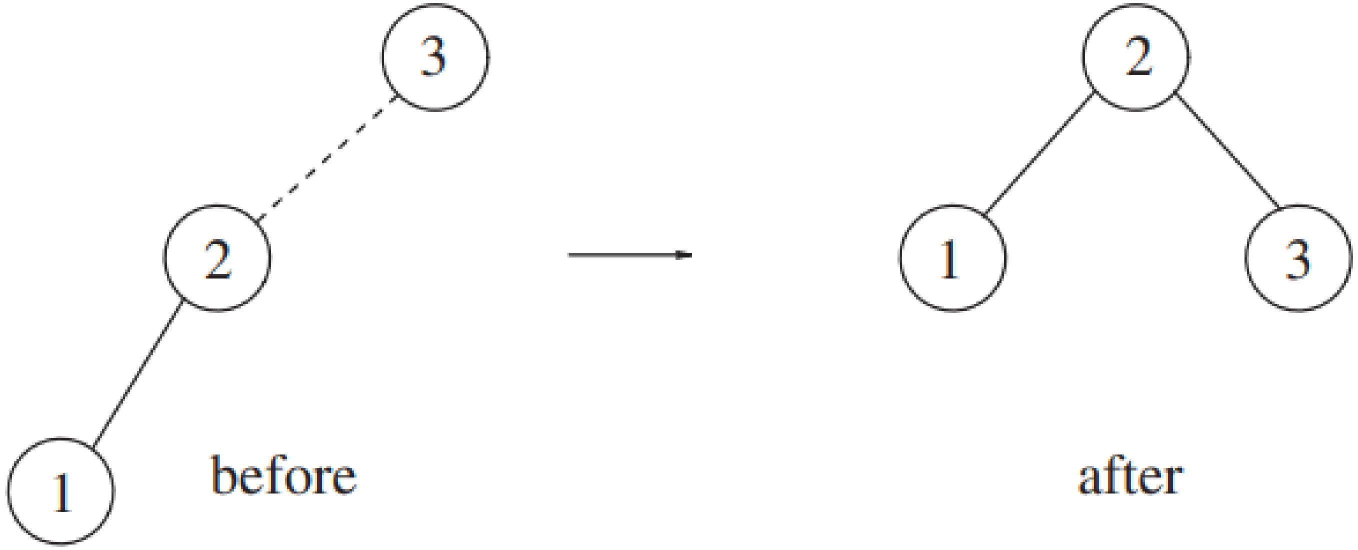


Tree rotations

- Each of the four cases is solved using a specific type of *rotation*
- Case 1: inserting into left subtree of left child of α solved by ***right rotation***
- Case 2: inserting into right subtree of left child of α solved by ***left-right rotation***
- Case 3: inserting into left subtree of right child of α solved by ***right-left rotation***
- Case 4: inserting into right subtree of right child of α solved by ***left rotation***

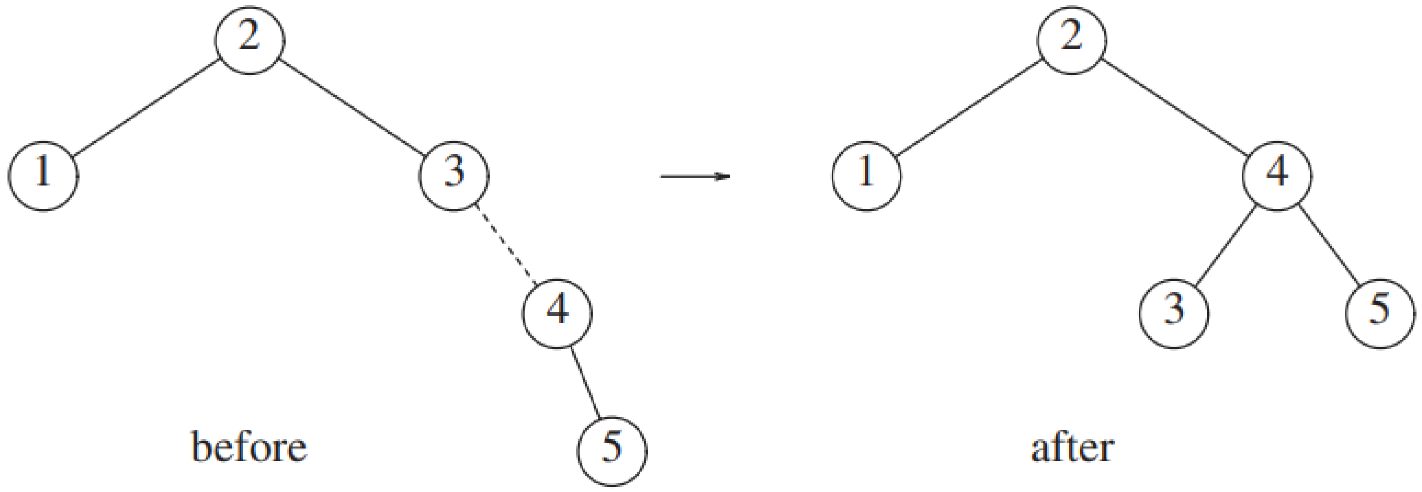
Single rotation (right rotation)

- Case 1: inserting into left subtree of left child of $\alpha = (3)$ solved by *right rotation*



Single rotation (left rotation)

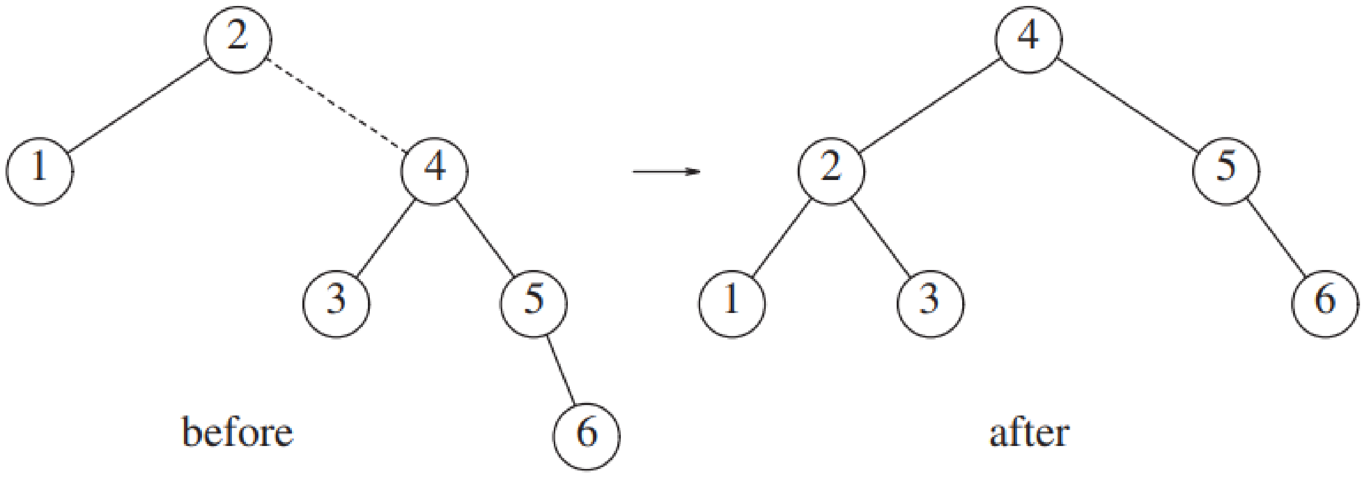
- Case 4: inserting into right subtree of right child of $\alpha = (3)$ solved by *left rotation*



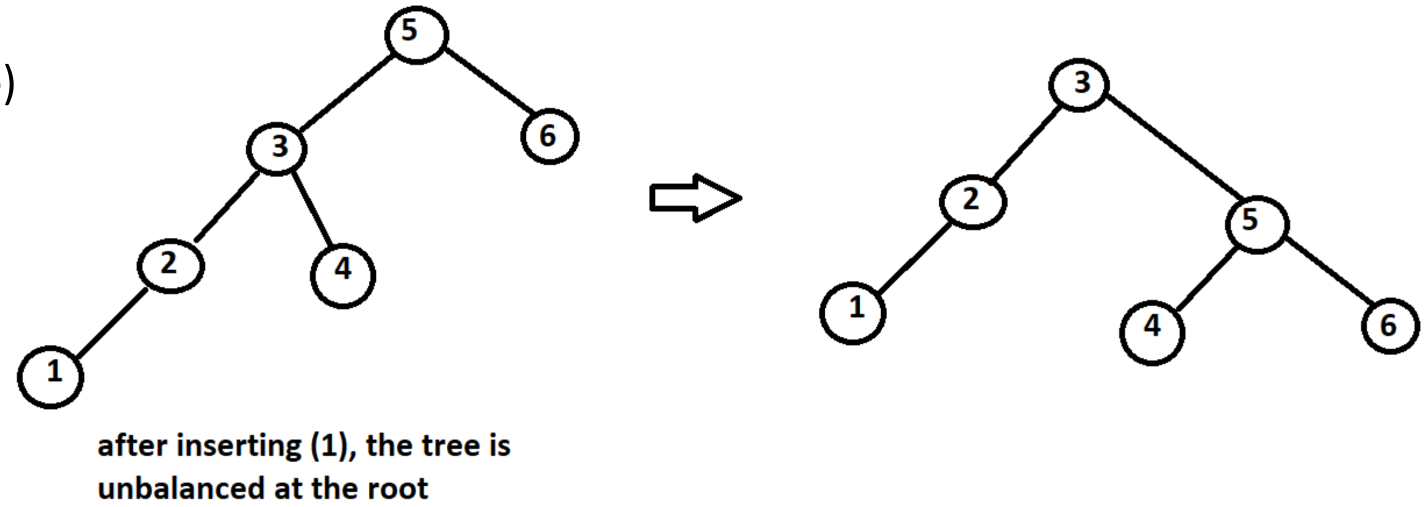
Single rotation with children

- Example: a) left rotation required (root is unbalanced) but new root (4) has a left child
- In left rotation, the left child of the node being rotated up (4 in this case) becomes the right child of the node being rotated down (2 in this case)
- Vice-versa for right rotation (b)

a)

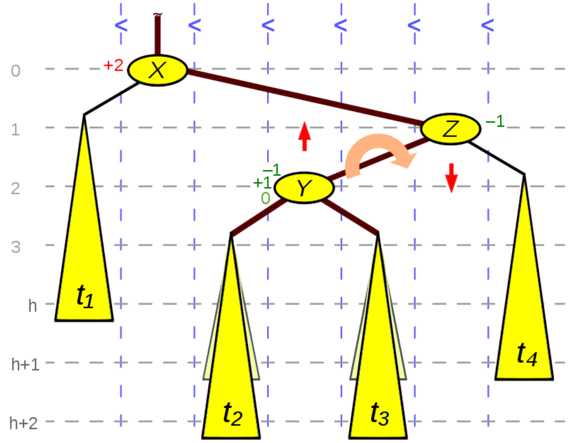


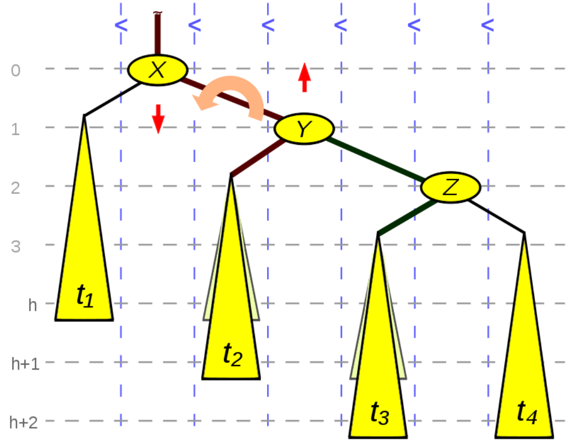
b)

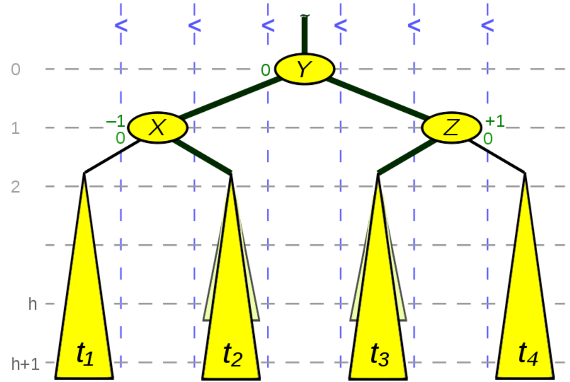


Double rotation

- Single rotation does not work for cases 2 or 3
- Example: one of node Z's left subtrees (t1, t2 or Y) is two levels deeper than X's left child (t1), making the tree unbalanced
- Single rotation will not solve this
- Fixed by *right left* rotation







Double rotation

- Before: inserting (15) as left child of (16) causes (7) to be unbalanced
- Case 3: inserting into left subtree of right child of $\alpha = (7)$
- Fixed by double rotation